Thresher: Automating the Unwrapping of Semantic Content from the World Wide Web

Andrew Hogue [1] [2]

[1] Google Inc.
New York, NY 10018
ahogue@google.com

David Karger [2]

[2] MIT CSAIL
Cambridge, MA 02139
{ahogue,karger}@csail.mit.edu

ABSTRACT

We describe Thresher, a system that lets non-technical users teach their browsers how to extract semantic web content from HTML documents on the World Wide Web. Users specify examples of semantic content by highlighting them in a web browser and describing their meaning. We then use the *tree edit distance* between the DOM subtrees of these examples to create a general pattern, or wrapper, for the content, and allow the user to bind RDF classes and predicates to the nodes of these wrappers. By overlaying matches to these patterns on standard documents inside the Haystack semantic web browser, we enable a rich semantic interaction with existing web pages, "unwrapping" semantic data buried in the pages' HTML. By allowing end-users to create, modify, and utilize their own patterns, we hope to speed adoption and use of the Semantic Web and its applications.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

General Terms

Algorithms, Human Factors

Keywords

Semantic Web, Haystack, RDF, wrapper induction, tree edit distance

1. INTRODUCTION

The Semantic Web promises to "bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users" [5]. Information currently prepared only for humans will be richly labeled, classified, and indexed, allowing intelligent agents to schedule our appointments, perform more accurate searches, and interact more effectively with the sea of data on the Web.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

These advances, however, rely on accurately converting and labeling data that currently exists only in human-readable form on the World Wide Web into a language that is easily manipulated by computers. The proposed semantic web language, RDF, can be used to record any desired relationship between pairs of objects. A *statement* is used to connect a pair of objects—referred to as the *subject* and *object*—with a particular *predicate* indicating their relationship. Choosing an appropriate collection of predicates and making statements enables detailed descriptions of objects using their classes and properties.

Ironically, much of the information presently available on the web is already *stored* in relational form, in the "deep web" of databases backing numerous web portals. The portals use various templating engines to transform their relational data into human-readable HTML documents. Unfortunately, this transformation obscures much of the (machine readable) relational structure in the process. The Semantic Web initiative relies heavily upon content providers to mark up their content with RDF, as they have easy access to the relational data behind the web pages, as well as the ability to alter the existing published content directly. Several tools, including browsers and distributed search engines for ontologies, have been developed with the explicit goal of making it easier for content providers to add semantic markup to their existing World Wide Web pages.

Current web publishers, however, have been slow to begin this process, as they often have little or no incentive to mark up their existing documents. Indeed, allowing agents to extract machine readable data directly from sites might obviate the need for users to actively visit those sites at all—an outcome advertising-driven providers might find unpalatable. As many end-user Semantic Web applications depend on the existence of useful RDF, its lack of adoption by content providers has also affected the utility of these applications.

In this work we present *Thresher*, a tool that allows *end-users* themselves, rather than content providers, to "unwrap" the semantic structures that have been buried inside human-readable html. Thresher extends existing web interfaces to give non-technical users the ability to easily "demonstrate" the extraction of semantic meaning from a web page. By watching these demonstrations, Thresher learns to extract analogous semantic content, either within the same page or from "similar" web pages. By giving users control over semantic content, we hope to reduce the dependence of

the Semantic Web on content providers and speed its adoption.

Thresher is aimed at sites that present many of the same type of object on a page or a site—such as movies at the Internet Movie Database, recipes at Epicurius, or stories at Slashdot. Such sites often create their web pages by feeding their relational data through a templating engine—essentially "filling in the blanks" of a common layout with item-specific content. Thresher takes the example of a user extracting the data about a particular object, and uses it to reverse engineer the templating engine, in order to extract corresponding information from web pages presenting objects with the same template.

Thresher allows users to indicate examples of semantic patterns, also called *wrappers*, simply by highlighting and marking up relevant features of a web page in a browser. From these positive examples, thresher induces a flexible, reusable pattern by applying a *tree edit distance* metric to find the *best mapping* between the example's and a target's *Document Object Models (DOMs)*—essentially their HTML parse trees. To provide feedback for the user, matches are displayed directly in the browser. Once a wrapper is created, the user can give it semantic meaning by indicating the *class* (type of object) extracted by the wrapper, as well as marking up certain portions as describing particular *properties* of that object.

To enable semantic interaction with web content, these predefined wrappers are reevaluated on subsequent visits to the same or similar pages. When matches are found, instances of the semantic classes represented by the pattern are created on the fly. These objects are underlaid in the browser, allowing us to reify the matched items into first-class objects, rather than flat text.

Thresher is able to "notice" and extract semantic content into RDF; it gains power when combined with a tool that can manipulate that RDF. By integrating the Thresher tool into the Haystack [18] information management environment, we gain the power to directly manipulate web page content *as semantic objects*. For instance, once a pattern is defined on a page listing upcoming seminars, a user can right click on any of the seminars on the page and add it to their calendar. Similarly, given a pattern that extracts individuals from an organization's membership page (such as that in the figures below), a right click can be used to add one of those individuals to an address book, or to send them email. In another instance, patterns defined on news sites effectively allow the user to create, modify, and subscribe to their own RSS feeds.

Although wrapper creation is relatively straightforward, many users may not wish to go through the effort to create these wrappers themselves. Because the wrappers created by Thresher are themselves RDF, it is easy to *share* them between users. One user creates the wrapper, sends its serialized RDF to another user, who simply installs the wrapper on their Haystack. One can imagine downloading a full set of wrappers for a given site and instantly enabling a full Semantic Web experience for users without the need for each user to author their own wrappers.

We begin in Section 2 by surveying existing work on wrapper induction, as well as interfaces for labeling and manipulating semantic information on the Web. To provide a visual framework for our discussion, we then describe the user interface for Thresher in Section 3. The underlying pattern induction algorithm and the methods for semantically labeling patterns are described in Section 4. We conclude by discussing experimental results in Section 5 and directions for future work in Section 6.

2. RELATED WORK

There are several existing Semantic Web efforts that focus on the process of content creation and semantic labeling. However, these projects have tended to focus on content providers and expert users, rather than allowing non-technical end-users to create and easily utilize their own metadata. Below, we consider three types of related work: tools that let users manually annotate html with semantic structure, tools that let developers hard-code extraction rules for certain types of semantic content, and tools like Thresher that learn extraction rules from examples.

2.1 Direct Annotation

The Annotea project [10] is a Web-based system, implemented using current web standards, that allows users to add descriptive metadata to individual pages. Users of an Annotea-enabled browser can create annotations and associate them with text within a given document, or with the page as a whole. Although annotations are described using RDF, and could, in theory, be used to create semantically labeled content, each annotation applies to a specific node in a specific page and must be manually created by the user. There is no way to generalize them to be applied to repetitive content. Similarly, the concept of Sticky Notes for the Semantic Web [11] has been proposed, but does not propose to generalize and automate the annotation process from examples.

2.2 Static Extraction Rules

The MIND SWAP project [8] has created a suite of tools aimed at giving users the ability to author, search, and browse Semantic Web documents. These include a program for semantically tagging delimited files, an editor for co-creating HTML and RDF documents at the same time, an interface for labeling non-text content with RDF, and an ontology manager for search and browsing. The most relevant tool to our work is the Web Scraper, which allows users to extract semantic information from structured HTML documents. To use the Web Scraper, a user must analyze the HTML source of a document and provide explicit delimiters for relevant information. Once the data has been extracted, an ontology browser is provided for semantic labeling. While the patterns created by the Web Scraper tend to be more powerful than those described here because of their explicit declaration, the interface for defining them is complex, requiring a knowledge of HTML and the low-level structure of a page. The Thresher system has been designed to be accessible to non-technical users, and allows pattern induction and utilization through a standard web browser interface, via existing well-understood operations such as highlighting and right clicking on a document.

Similar to Thresher, the news extraction approach presented by Reis, et. al. [19] utilizes tree edit distance to find similarities between web pages containing semantic content. This system takes a highly domain-specific approach, crawling and clustering news web sites using predefined structural properties of those sites. A restricted tree edit distance algorithm is then used on all pages within each cluster to generate a pattern. Several heuristics are then used to au-

tomatically label the title and body of each news article. This approach demonstrates the power of tree edit distance for forming reusable patterns on web pages, correctly extracting news from 87.71% of the sites surveyed. However, where Thresher relies on the user to provide semantic labels for their selections, this news extraction system relies on several hard-coded heuristics and rules to complete the labeling process. Thresher's end-user approach also allows content to be extracted from a wide range of sites without domain-specific code.

The XPath standard [3] is another useful tool for extracting data from hierarchical documents such as XML and HTML, and several tools such as xpath2rss [17] have been built with it. Similar to the Web Scraper, though, these require the user to have a detailed knowledge of the document and of XPath's language for describing patterns. Few tools have been developed to allow intuitive, interactive induction of useful patterns.

Magpie [6] is a tool for layering semantic information about known entities over web documents. Drawing from a database of predefined people, projects, organizations, and other concepts, Magpie highlights these entities on a web page at the request of the user. New context menus are created which allow the user to view the connections between these entities. Thresher and Magpie are similar in their approach to overlaying semantic information on web pages, and allowing users to interact with that information. Because of its embedding in the Haystack environment, Thresher allows more complex interaction with the semantic information found in web pages (for example, emailing a person whose information is listed on the web). Thresher also allows the user to create wrappers and imbue them with semantic meaning, rather than drawing its content from a predefined database.

2.3 Learning Extraction Rules

One interactive system that enables users to iteratively develop reusable patterns in various types of documents is LAPIS [15]. Patterns are developed using a language called *text constraints*, specifying operators such as *before*, *after*, *contains*, and *starts-with*. By using a pre-defined library of parsers that tokenize and label the document, users can create patterns of arbitrary complexity, or allow the system to infer them from examples. Currently, users may utilize LAPIS to perform such tasks as simultaneous text editing and outlier finding; to date it not been applied to the Semantic Web.

FlipDog[1], developed by Tom Mitchell at WhizBang! Labs and now owned by Monster.com, is a commercial system for automating the extraction of semantic data from the web. FlipDog was trained to locate information from job postings on corporate sites, including the employer's name, the position, the salary, the job's description, and other common fields. This information was then extracted into a central repository of job postings, allowing it to be easily searched and correlated.

The Daily You [21] is an automated classification tool which attempts to classify which portions of a page are "interesting" to the user and which are not. The system is based on the concept that nearby portions of a web page's DOM tree are more likely to be related than distant ones. By observing users as they browse a site, The Daily You is able to classify and remove irrelevant parts of the page like ads and navigation boxes, while highlighting links that might be of particular interest to the user. This tree-structured approach directly influenced Thresher's heuristics for creating wrappers.

Wrapper induction is defined by Kushmerick [13] as the task of learning a procedure for extracting tuples from a particular information source from examples provided by the user. Kushmerick defined the HLRT class of wrappers. These wrappers were restricted to locating information that is delimited by four types of flags, the "head," "left," "right," and "tail." Subsequent work on wrapper induction involves hierarchical structure [16] and probabilistic models [7, 20]. Our work builds on the conceptual framework of wrapper induction, extending it with a pattern inference and matching framework specialized for the DOM (hierarchical document object model) model of web documents.

Our work embeds in the Haystack [18] information management client. This system has strong ties to the Semantic Web, in that it is based on the RDF standard [2] and incorporates many standard ontologies, including the Dublin Core [1] and the Simile project [4]. Through these ontologies, users of Thresher are provided with an existing base of classes and properties to assign to wrappers. Another key benefit of the Haystack interface is that every object is semantically "alive" to the user. This means that Thresher can provide relevant context menus for any element displayed on the screen. For instance, in the interface for composing an email message the "To" and "CC" fields not only provide context menus for adding additional recipients, but also provide menus to interact with the actual "Person" objects behind existing recipients. These semantically-driven context menus provide a simple, intuitive interface for non-expert users to create, manage, and use semantically structured data on the Web.

3. USER INTERFACE

The Thresher interface provides four main functions to the user, described in more detail below:

1. *wrapper creation*, from a highlighted example
2. the specifying of *additional examples* for an existing wrapper
3. *semantic labeling* of wrappers with RDF properties
4. *interaction* with labeled content, via semantic context menus

The process of *wrapper creation* begins when the user navigates to a page containing semantic content in the Haystack web browser. Because the Thresher system is "always on" in the Haystack browser, creating a wrapper is initiated simply by highlighting the relevant semantic content on the page. The user then right-clicks and chooses "Create a Wrapper" from the context menu that appears. They are then prompted to provide additional information necessary for wrapper creation: the semantic class of the selected object (what "kind of thing" is being described by the page fragment) and a name for the wrapper (optional, but useful so that it can be located and modified later). Figure 1 shows these two steps on the MIT CSAIL faculty page.[2] Here

[1] `http://flipdog.monster.com`

[2] http://www.csail.mit.edu/biographies/PI/biolist.php

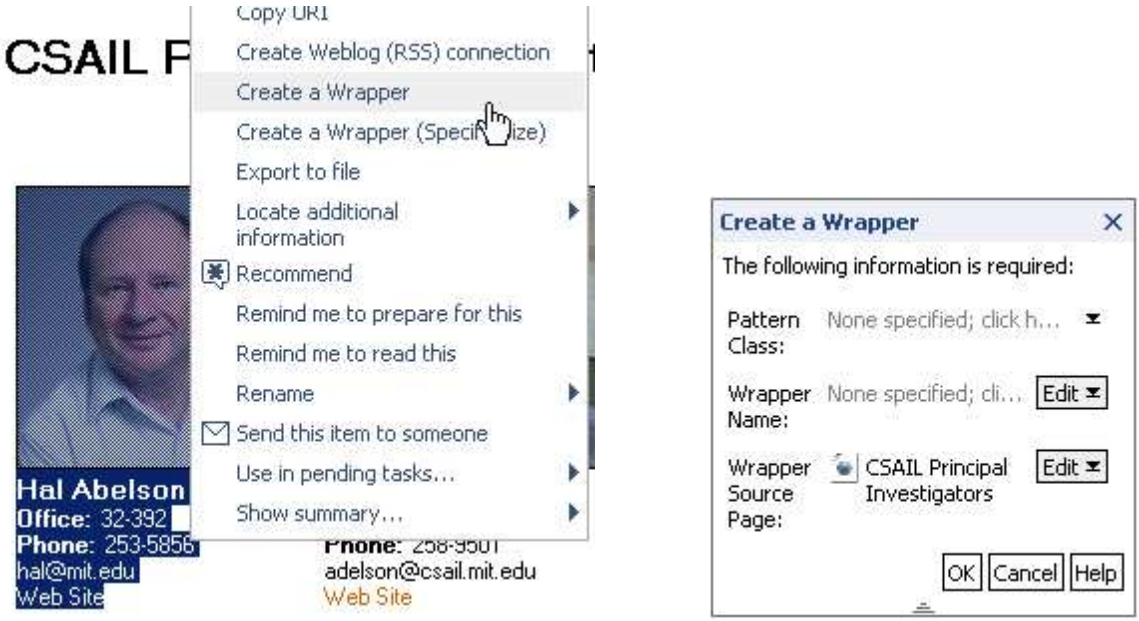


Figure 1: Creating a wrapper on the CSAIL Faculty page.

we are creating a wrapper to match each individual faculty member, so the corresponding semantic class is PERSON.

Once the user confirms their choices, the wrapper induction algorithm (described in the next section) runs in the background and creates a generalized wrapper from the first selection. The wrapper is then matched back against the same page, and each match is highlighted to provide visual feedback for the user. This highlighting is shown in Figure 2.

While semantic objects may often be unwrapped using a single example, there are cases where more than one example is necessary. Thus, an interface is provided to allow the user to add *additional examples* to an existing wrapper. To do this, the user simply highlights another example on the page, right clicks, and chooses "Add an Example to a Wrapper" from the context menu that appears. They are then prompted to select the wrapper to modify, and the system integrates the new example into the wrapper.

The ability to add additional examples is essential for pages without enough information on a single page to create an effective wrapper—for example, when only one instance of the target object is present on any one page. In these cases, the user may need to provide the first example on one page, and an additional example on another page. The mechanism for adding additional examples described above works well for these cases - the user selects the additional example on the new page and chooses the existing wrapper from the list that appears.

Once the user is satisfied with the matches the wrapper provides, they may begin *semantic labeling* of the pattern, marking up the relationships described in the html fragment. To do this, the user selects a portion of one of the matches on a page that represents a semantic property. They then right-click to bring up the context menu and select "Add a Property to this Wrapper." The user is then provided with a dialog in which they are asked to select from a list of properties that are applicable to the class assigned to the given wrapper. This list is generated by inspecting the schema associated with the type of object being unwrapped, and determining which predicates apply to that type of object (more formally, we query the schema to find out which RDF predicates have the current wrapper's semantic class as their `rdfs:domain`). So, for example, when the type of object is PERSON, then available predicates include NAME and ADDRESS. General properties that apply to all classes, such as `dc:title`, are also listed. Once the user has selected a property, it is bound to the wrapper as described in the next section. An example of this process, and the visual feedback given to the user after the pattern is updated, is shown in Figure 3,

Finally, once a wrapper is fully designed, Thresher allows users to *interact* with web content in a fully semantic way. Every time a user browses to a page with a wrapper, we execute the matching algorithm for the wrapper. Any elements in the document that match the pattern are "underlaid" with dynamically generated semantic objects. These objects are fully-functional semantic instances, with properties supplied by the RDF predicates assigned during wrapper induction. Because Haystack provides content-specific context menus for semantic data, the user may now interact with semantic content in the web page as if it were a first-class RDF object.

For example, in Figure 4, the user has right-clicked on a faculty member in the CSAIL directory. Because a `Person` semantic wrapper has been defined for this page, the user is presented with a context menu relevant to that class. This includes such items as "Remind me to contact this party" and "Compose Email Message." Using properties of these objects drawn from the page, commands like "Compose Email Message" will be passed the appropriate information (in this case, an email address) to execute their actions.

4. PATTERNS

The user interface described in the previous section makes it easy for users to label semantic data on the Web, and



Figure 2: Feedback during wrapper creation by highlighting matched elements.

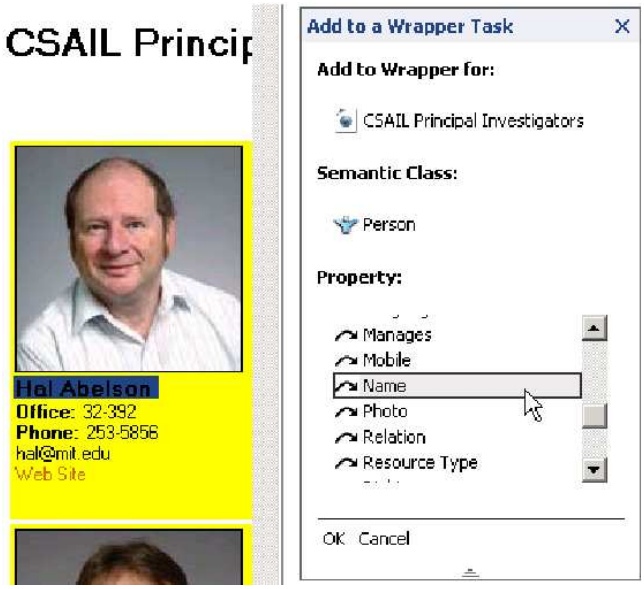


Figure 3: Adding a property to a wrapper.

later revisit that data and interact with it. These actions are supported by wrapper induction and semantic labeling algorithms that take advantage of the hierarchical structure of the DOM of a web page.

4.1 Wrapper Induction

We begin by describing our algorithm for learning a pattern from user-provided examples. Because of the way HTML is rendered in a browser, when a user selects a contiguous block of a web page as an example, they are, in effect, selecting a *subtree* from the page's DOM. Our postulate is that instances of the target semantic object are rendered into a particular shape of subtree. Some portion of that subtree is the actual content of the object being viewed, while another portion reflects the "layout" of that content. For repeated, semantic objects, we expect to see the general



Figure 4: Interacting with an existing wrapper on the faculty directory page.

layout repeated, while we expect the content to vary for each instance.

Given a set of example subtrees, our task is to distinguish "content bearing" subtrees from all other HTML on the page, and then to distinguish the layout portion of those trees from the content portion. Our approach to both problems is to be maximally conservative: we look for a pattern that all the examples fit, but that generalizes as little as possible subject to that constraint. Intuitively, we expect that some part of each example is exactly the same, that this repeating identical portion is sufficient to identify occurrences of the object, and that it represents the (unvarying) layout portion of the pattern. This repetition is generally a result of the way in which pages are generated. When a content provider wishes to display a large number of related items, each of the same semantic class, they tend to automate the process. As a result of this automation, each instance of that semantic class on the page tends to have the same basic HTML structure, "filled in" with different text content.

In practice, content sometimes influences layout. Thus, we cannot simply declare that all object presentations must have identical layouts—we need to introduce some flexibility. In order to identify the repeating elements, we attempt to "align" pairs of examples, matching up corresponding elements of each subtree. Those elements that can be matched up are presumed to be (unvarying) layout, while those that do not match are taken to be (variable) content. In order to perform the alignment, the best mapping is defined as the mapping between the nodes of two trees with the lowest-cost *tree edit distance* [22].

Once we have found this best mapping, we create our pattern by starting with one of the examples and replacing any nodes that do not recur (match) in the other examples with *wildcard* nodes. This creates a "skeleton" tree that contains only the common nodes among all examples. For example, Figure 5 shows the best mapping between two `TalkAnnouncement` subtrees, as well as the resulting pattern. The best mapping procedure preserves those DOM

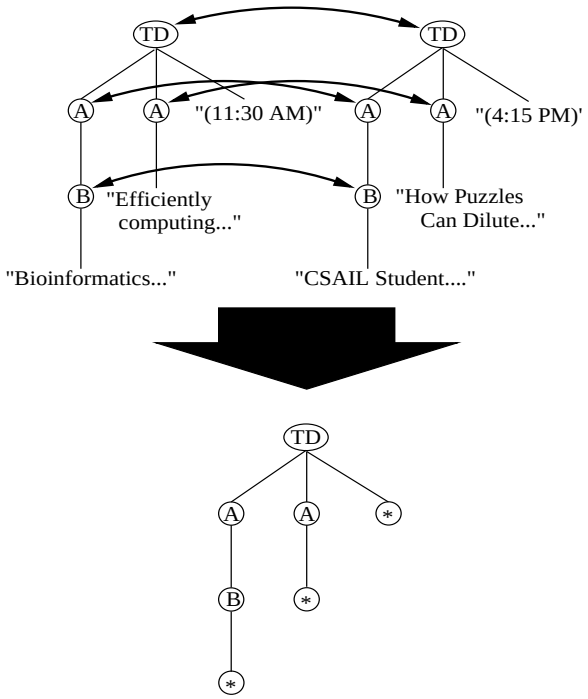


Figure 5: Generating a wrapper from the best mapping between two example trees.

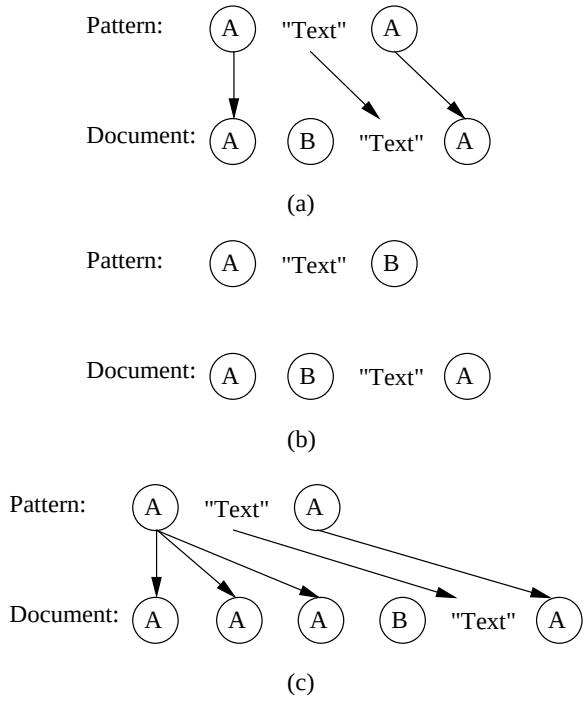


Figure 6: Examples of aligning two lists of child nodes. (a) shows a valid alignment. (b) shows a pair with no valid alignment. (c) shows a valid alignment using the list heuristic.

elements that are in common between the examples, while discarding those that exist only in one example or another. In this way, Thresher "reverse engineers" the scripts that created the page, separating the common layout structures they produce from the variable semantic content that they fill in.

Once the general pattern is created, we can match it against the document by simply looking for subtrees on the page that have the same structure. First, we find nodes in the document with the same tag as the root of the pattern. For each of those nodes, we attempt to *align* the children of the document node against the children of the pattern. A valid alignment is one in which each pattern node maps to at least one document node. This process continues recursively until every node in the pattern is bound to a node in the document, resulting in a match, or we fail to find a binding.

In more detail, we begin by trying to align the root, $P[0]$, of the pattern with each node in the document to be matched, T. If we find a node v such that $P[0]$ and v match, we recurse, attempting to align the children of $P[0]$ with the children of v. We take the list of children of $P[0]$ and of v, and attempt to find an alignment. An alignment is a mapping from the children of $P[0]$ to the children of v where every child of $P[0]$ maps to at least one child of v, and if $P[q]$ maps to $T[r]$ and $P[s]$ maps to $T[t]$, and $P[q]$ is a left-sibling of $P[s]$, $T[r]$ is also a left-sibling of $T[t]$. That is, sibling order is preserved in the alignment.

Figure 6(a) shows a valid alignment between the a list of pattern children and a list of document children. Figure 6(b) shows a pattern-document pairing with no valid alignment. Note that the `''Text''` and `B` nodes do *not* align in Figure 6(b) because sibling order must be preserved.

4.2 A List Heuristic

A common occurrence led us to introduce an important heuristic. If the user's selection contains a set of similar, repeating elements (for instance, rows of a table or an HTML list), the wrapper algorithm as described above will create a pattern with the exact number of repeated elements in the example. For instance, a table with three rows will create a pattern with three rows. In many instances, however, a semantic object may incorporate a variable length *list* of relationships. For example, a recipe might list a variable number of ingredients, a movie a variable number of actors in the cast, or a news site a variable number of news articles under a given heading. Trees representing such records with different-length contents do not align well under the standard edit distance metric.

List collapse deals with this variation by collapsing highly-similar neighboring elements into a single wildcard node. During matching, this wildcard is then allowed to match multiple elements. An example of this collapse is shown in Figure 7, where a set of anchor tags is collapsed into a single wildcard node. Later, any number of links may be matched by this wrapper, provided they fall within a `TD` and are preceded by the text "Links:". An example of this alignment matching step is shown in Figure 6(c).

To determine when neighboring trees are sufficiently similar to be collapsed, we use the same edit distance metric as we use to match full patterns. When the edit distance between two adjacent subtrees is sufficiently low, we collapse them. Our measure of similarity is the *normalized cost*, $\hat{\gamma}$,

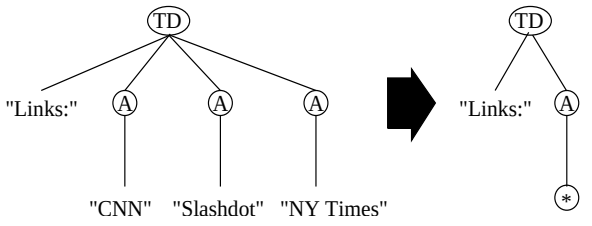


Figure 7: Generating a wrapper with list collapse.

of the edit distance between the two subtrees, defined as:

$$\hat{\gamma}(T_1 \to T_2) = \frac{\gamma(T_1 \to T_2)}{|T_1| + |T_2|},$$

where $\gamma(T_1 \to T_2)$ is the cost (edit distance) of the best mapping between subtrees T_1 and T_2. Because the most expensive mapping involves deleting all of T_1 and inserting all of T_2, and we charge a cost of 1 per node inserted or deleted, we also know that

$$0 \leq \hat{\gamma}(T_1 \to T_2) \leq 1$$

for any mapping $T_1 \to T_2$. This normalization of the cost allows us to set an absolute cost threshold for the list collapse heuristic described above.

4.3 Finding More Examples

The above heuristic "coped with" lists; in another heuristic we take advantage of them. To reduce user workload, we would like to minimize the number of examples a user needs to provide in order for the system to learn a useful pattern. To provide additional example subtrees without asking the user for another selection, we look to the nodes neighboring (with the same parent as) the root of the user's selection. For each neighboring subtree, we compute the normalized cost of the best mapping between it and the original selection. If this cost is less than some threshold, we automatically consider it as an additional example for the wrapper.

Again, we justify this heuristic by noting that semantic content often appears in "list" form on the web, with multiple instances of the same type of content appearing in neighboring nodes of the DOM tree. Thus it is a plausible assumption that nodes neighboring the root of the user's selection are also instances of the same semantic type. This assumption has been born out by experimental trials. By using a tree edit distance threshold to weed out nodes, we help to ensure that this heuristic does not pick up spurious examples.

4.4 Semantic Labeling

Once a wrapper has been created, Thresher must provide a means for applying semantic meaning to it. When we created the patterns, we began by taking a single, specific instance, then generalized it by mapping it to other instances, removing nodes that the instances did not have in common. In semantic terms, what we did by removing these specific instance nodes was map the instances into a generic description of the structure of the semantic class they represent. Because the pattern itself is a general description of a semantic class, we simply bind the class to the *entire* pattern—i.e., we posit that any html matching the pattern represents an instance of the given class.

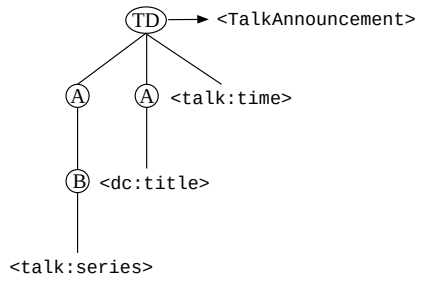


Figure 8: Labeling a wrapper with semantic meaning.

In addition, when we found nodes that differed between examples, our pattern induction algorithm changed these nodes into *wildcards*. Because these wildcards provide a direct mapping between the wrapper's structure and the variable features contained in the web page's structure, they make a natural binding location for our semantic properties. Thus, when the user selects a part of the DOM and applies a semantic property to it during the wrapper creation process, we simply bind the RDF predicate representing the property to the selected node. Figure 8 shows the pattern of Figure 5 now labeled with a semantic class and with properties bound to its wildcard nodes.

The matching process now provides a means to extract structured semantic content from a web page. Each time we find a match to our pattern, we create an instance of the semantic class it represents. The wildcards in the pattern (which resulted from removing unmapped nodes) are bound to nodes in the document. If the user has labeled these slots with RDF predicates, the matched text is used to assign the properties of the new instance. This instance may then be supplied to the browser or to an agent to be used as first-class semantic content.

5. EXPERIMENTAL RESULTS

The development of the Thresher wrapper induction system was based on a survey of popular web sites containing a variety of semantic information.[3] Table 1 gives a brief summary of our results on some of these sites.

Overall, our experiments validate our hypothesis that edit distance can create flexible patterns from few examples. On numerous sites, as few as one or two examples are enough to create a useful pattern. For example, Figure 9 shows the wrapper that was induced for the `SearchResult` class on `http://google.com`. We were able to create this wrapper from a single example on the search results page by gleaning additional examples from the selection's neighboring nodes, as described in Section 4.3.

Figure 10 shows the wrapper for the `Actor` property on `http://imdb.com`, an interesting example of the effectiveness of our *list collapse* heuristic. This pattern was also created with a single example by highlighting one cast member. The list collapse heuristic then merged the `TR` nodes into a single pattern node, which matches every cast member in the list. It is interesting to note that the subtree containing the words "Cast overview, first billed only:" was *not* collapsed, despite having the same tag name at its parent node. This

[3]For a complete listing of these sites and the related experimental results, see the first author's Masters thesis [9].

Site	URL	Semantic Class	Examples Required	Comments
Google	http://google.com/search	`SearchResult`	1	Context heuristic found more examples
Yahoo!	http://yahoo.com/search	`SearchResult`	1	Context heuristic found more examples
Weather.com	http://weather.com	`LongRange Forecast`	2	List collapse
IMDB	http://imdb.com/title	`Actor`	1	Context heuristic found more examples
IMDB	http://imdb.com/title	`Director`	2	Examples on multiple pages
IMDB	http://imdb.com/title	`Writer`	2	Examples on multiple pages
IMDB	http://imdb.com/title	`Movie`	fail	Could not wrap full-page class
Slashdot	http://slashdot.org	`StoryIcon`	1	
Slashdot	http://slashdot.org	`StoryPoster`	1	
Slashdot	http://slashdot.org	`StoryLink`	2	
CSAIL Directory	http://www.csail.mit.edu/biographies/PI/biolist.php	`Person`	2	Second example necessary for faculty without a web page
CSAIL Event Calendar	http://www.csail.mit.edu/events/eventcalendar/calendar.php	`Talk Announcement`	2	Second example necessary for talks not in a series
MIT Course Catalog	http://student.mit.edu/catalog/*	`Course`	1	
Mozilla Bugzilla	http://bugzilla.mozilla.org/show_bug.cgi	`BugStatus`	2	Examples on multiple pages
ESPN MLB Scoreboard	http://espn.com/mlb/scoreboard	`BaseballGame`	5	Extra examples necessary due to multiple slots in box scores
Java API Reference	http://java.sun.com/j2se/1.4.2/docs/api/index.html	`JavaClass`	fail	Could not wrap full-page class
Java API Reference	http://java.sun.com/j2se/1.4.2/docs/api/index.html	`Method`	2	Second example necessary for variable number of arguments
50 States	http://50states.com/*	`StateCapital`	2	Examples on multiple pages
EBay	http://cgi.ebay.com/ws/eBayISAPI.dll	`AuctionTitle`	2	Examples on multiple pages
EBay	http://cgi.ebay.com/ws/eBayISAPI.dll	`Auction StartingBid`	2	Examples on multiple pages
Barnes & Noble	http://search.barnesandnoble.com/booksearch/isbnInquiry.asp	`Book`	fail	Could not wrap full-page class
Barnes & Noble	http://search.barnesandnoble.com/booksearch/isbnInquiry.asp	`BookTitle`	2	Examples on multiple pages
Barnes & Noble	http://search.barnesandnoble.com/booksearch/isbnInquiry.asp	`BookPrice`	2	Examples on multiple pages

Table 1: Number of examples necessary to form a wrapper.

subtree had a higher edit distance cost, and because of this our algorithm correctly inferred that it did not contain the same type of semantic content. Instead, this subtree serves as a "flag" that allows our pattern to match only the list of actors and exclude other elements that do not begin with the text "Cast overview..."

Despite these and other successes, there are several sites where we either failed to induce a wrapper, the wrapper was incorrect, or generating a valid wrapper took numerous examples. Several of these failure modes include:

Full-page Classes Many of the semantic classes we examined were "full-page." On `http://imdb.com`, for example, the entire page represents a single instance of the `Movie` class. Because its running time is $O(n^2)$, performing the tree edit distance calculation on entire pages was prohibitively expensive, and we could not create wrappers for this information.

Selection Inconsistencies Our system depends on reliably extracting the user's selection from the browser and mapping this to the related subtree in the page's DOM. In several cases, bugs in the web browser itself[4] prevented this, resulting in failed wrappers.

Large Numbers of Semantic "Slots" Wrappers with a large number of wildcards often take a large number of examples to generalize properly, and then take a long time to match, as wildcards provide many more alignment opportunities than non-wildcard elements. One example of this is the `BaseballGame` class on the ESPN site, where each inning for each team needed to be generalized and matched separately, creating a pattern with more than 18 wildcards.

[4]Thresher has been implemented to work with both Microsoft Internet Explorer and Mozilla.

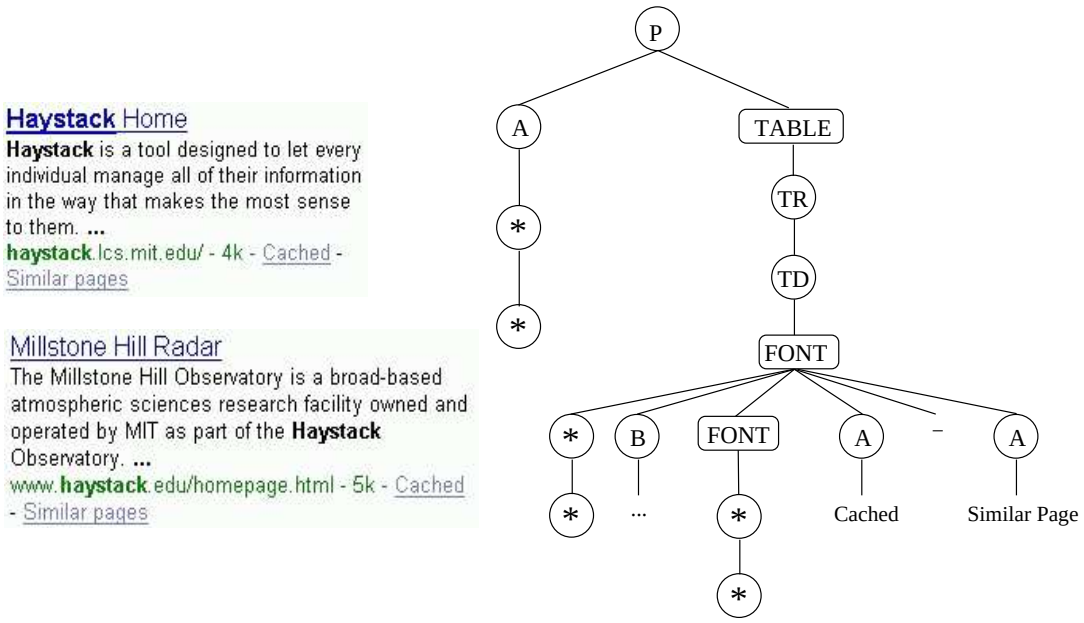


Figure 9: The `SearchResult` wrapper on `http://google.com`.

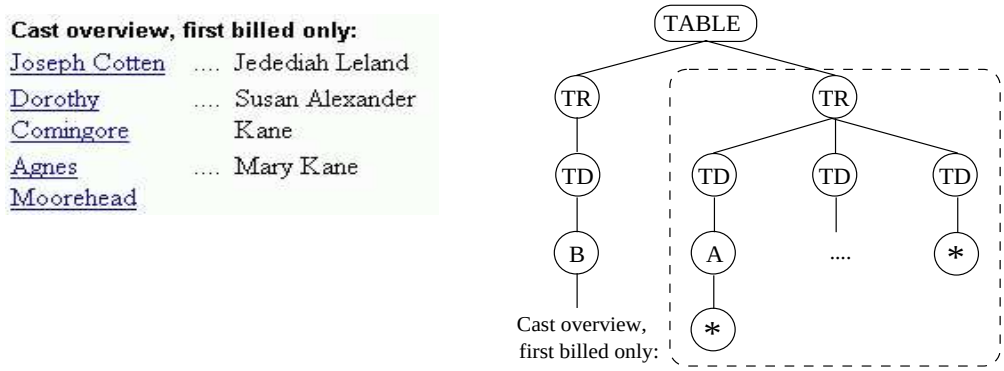


Figure 10: The `Actor` wrapper on `http://imdb.com`.

6. FUTURE WORK

In addition to the user interface and algorithmic ideas described here, several other future improvements have suggested themselves:

Document-level Classes As mentioned in Section 5, there are many times when an entire web page represents a single semantic class, with items on the page detailing the properties of that class, as with the `Movie` class on `http://imdb.com`. We would like to allow for applying RDF statements that tie predicates on a page to the page-level class, without running the $O(n^2)$ tree edit distance algorithm on the entire page.

Labeling Across Pages Much semantic information transcends page boundaries. For instance, on the CSAIL events calendar, only the talk's `series`, `title` and `time` are listed on the calendar page, while the `speaker`, `date`, and `abstract` are only available by clicking on the title link. We would like to develop a system that allows semantic classes and properties to span multiple pages.

Negative Examples There are cases where our system creates wrappers that are too general in nature based on the positive examples provided by the user. We would like to allow the user the ability to make wrappers more restrictive by giving *negative* examples.

Natural Language Extraction The wrappers generated by Thresher can only capture classes and properties that are defined by nodes in the DOM tree. By applying natural language techniques, such as part-of-speech tagging or parse trees, our edit distance techniques might be extended to reach into the raw text at the leaves of the DOM.

Wrapper Verification Web pages are constantly in flux, and methods for validating wrappers are important [12]. We would like to develop an efficient way to verify that the semantic content being returned by the wrappers is still accurate.

"Push" Wrappers The wrappers defined here are laid out in a context of *pulling* information off of the web. How-

ever, many sites work both ways, also allowing the user to fill out forms or submit other types of information. These form entries also have semantic types associated with them, such as a `Person` class, with properties such as `name`, `address`, and `email`. Manuel [14] has expanded the ideas presented in Thresher to learn how to perform these "web operations" using data in the Haystack repository.

Agent Interaction Once defined, our wrappers can reliably extract semantic information from web pages, even independent of user interaction. This makes them ideal for interfacing with autonomous user agents. For example, if a user defines a wrapper using the `News` ontology, an agent that aggregates all of the user's news feeds into a collection could notice this and automatically integrate content from the new wrapper.

7. CONCLUSION

In this paper we have described Thresher, a system that gives non-technical end-users the ability to describe, label, and use semantic content on the World Wide Web. Previous work on labeling content on the Semantic Web has always focused on either content providers (in the form of page authoring tools) or on technically proficient end-users who know HTML and RDF. The tools described here rely on simple interfaces and user actions already present in existing web browsers, such as highlighting and right clicking on content.

In addition, we have provided a powerful algorithm for creating patterns from tree-structured data using the edit distance between examples. Along with several heuristics to improve its efficiency and accuracy, this method allows us to create reliable patterns with as little as a single example of the relevant content.

The wrappers created by Thresher create an important bridge between the *syntactic* structure and the *semantic* structure of the web page. In general, this parallel structure has always existed, abstractly, in the intentions of the page's creator and in the interpretations of the page's reader. In our system, however, the act of building a wrapper for this content makes the connection explicit on the user side. It is from this syntactic-semantic bridge that our wrappers get their power.

8. REFERENCES

[1] Dublin core metadata initiative. http://purl.org/metadata/dublin_core, 1997.

[2] Resource Description Framework (RDF) specification. http://www.w3.org/RDF, 1999.

[3] XML Path language (XPath) specification. http://www.w3.org/TR/xpath, 1999.

[4] Simile: Semantic Interoperability of Metadata and Information in unLike Environments. http://simile.mit.edu, 2004.

[5] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):35, May 2001.

[6] J. Domingue, M. Dzbor, and E. Motta. *Handbook on Ontologies in Information Systems*, chapter Semantic Layering with Magpie. Springer Verlag, 2003.

[7] D. Freitag and A. McCallum. Information extraction with HMM structures learned by stochastic optimization. In *AAAI/IAAI*, pages 584–589, 2000.

[8] J. Golbeck, M. Grove, B. Parsia, A. Kalyanpur, and J. Hendler. New tools for the semantic web. In *Proceedings of 13th International Conference on Knowledge Engineering and Knowledge Management*, Oct 2002.

[9] A. Hogue. Tree pattern inference and matching for wrapper induction on the World Wide Web. Master's thesis, Massachusetts Institute of Technology, May 2004.

[10] J. Kahan and M.-R. Koivunen. Annotea: an open RDF infrastructure for shared web annotations. In *World Wide Web*, pages 623–632, 2001.

[11] D. Karger, B. Katz, J. Lin, and D. Quan. Sticky notes for the semantic web. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, pages 254–256, 2003.

[12] N. Kushmerick. Wrapper verification. *World Wide Web*, 3(2):79–94, 2000.

[13] N. Kushmerick, D. S. Weld, and R. B. Doorenbos. Wrapper induction for information extraction. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.

[14] R. Manuel. Learning the process of World Wide Web data retrieval. Master's thesis, Massachusetts Institute of Technology, January 2005.

[15] R. C. Miller and B. A. Meyers. Lightweight structured text processing. In *Proc. of USENIX 1999 Annual Technical Conference*, pages 131–144, Monterey, CA, USA, June 1999.

[16] I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In O. Etzioni, J. P. Müller, and J. M. Bradshaw, editors, *Proc. of the Third International Conference on Autonomous Agents*, pages 190–197, Seattle, WA, USA, 1999. ACM Press.

[17] M. Nottingham. xpath2rss HTML to RSS scraper. http://www.mnot.net/xpath2rss/, 2003.

[18] D. Quan, D. Huynh, and D. R. Karger. Haystack: A platform for authoring end user semantic web applications. In *Proc. 2nd International Semantic Web Conference*, 2003.

[19] D. C. Reis, P. B. Golgher, A. S. Silva, and A. F. Laender. Automatic web news extraction using tree edit distance. In *Proceedings of the 13th International Conference on the World Wide Web*, pages 502–511, New York, NY, 2004.

[20] K. Seymore, A. McCallum, and R. Rosenfeld. Learning hidden Markov model structure for information extraction. In *AAAI 99 Workshop on Machine Learning for Information Extraction*, 1999.

[21] L. K. Shih and D. Karger. Using URLs and table layout for web classification tasks. In *Proceedings of the 13th International Conference on the World Wide Web*, pages 193–202, New York, NY, 2004.

[22] K.-C. Tai. The tree–to–tree correction problem. *J. Association of Computing Machinery*, 26(3):422–433, July 1979.